Nonlinear Network Programming on Vector Supercomputers

Stavros A. Zenios
John M. Mulvey

Report EES-85-13

Engineering-Management Systems
Civil Engineering Department
Princeton University
Princeton, NJ 08544

ABSTRACT

The parallelism built in vector supercomputers raises several challenging issues for designers of optimization algorithms. We survey recent trends in parallel computer systems, studying the impact of vector computing on nonlinear network programming. A general framework for vectorizing optimization software is proposed, and applied in the context of two nonlinear network codes: (1) a primal truncated Newton (NLPNETG) method and (2) a simplicial decomposition (GNSD) method. Computational experiments on a Cray X-MP/24 system -- testing the nonlinear network codes and comparing the results with a general purpose optimizer MINOS -- are included.

Keywords: Vector computers; Network optimization; Nonlinear programming; Vectorization; CRAY X-MP; CRAY Fortran

February 15, 1986

This research was funded in part by NSF grant DCR-8401098 and NASA grant NAG-1-520.

Nonlinear Network Programming on Vector Supercomputers

Stavros A. Zenios
John M. Mulvey

Report EES-85-13

Engineering-Management Systems
Civil Engineering Department
Princeton University
Princeton, NJ 08544

1. Introduction

The first computers were designed to satisfy the computational needs of scientists and engineers. Once the community began operating these computers, they realized that problems existed whose size exceeded the capabilities of these machines -- maybe an indication that human vision outperforms human engineering. As a direct result of the existing need for bigger and faster machines the first supercomputers were built -- those being nothing more than the most powerful computer at any point in time.

Improvement in hardware performance of computers and supercomputers has been dramatic[4]. First came the change from bulky and sensitive vacuum tubes to the small and reliable semiconductor transistor. This was followed by a twenty year period of packing hundreds, to hundreds of thousands of transistors on a silicon chip. These developments were the major factor behind improved computational performance, and the basic computer logic design did not change much - if it changed at all - from the original von Neumann prototype.

The improved performance of the silicon chip is, however, reaching its (quantum-mechanical) limits. In addition, with operating gate speeds of the order of few nanoseconds, a limit is imposed by the distance a signal can propagate: at 9.5 nsec for the CRAY X-MP the limit is roughly approximate to the linear dimensions of the computer. Thus, greater computer performance can be achieved only through radical changes in computer architecture; today's supercomputers are shifting from the uni-processor design to architectures with many processors that per-

form independent operations concurrently.

This trend in changing computer architecture presents the algorithmic designer with new challenges. In the context of mathematical programming algorithms, there have been a number of theoretical studies in unconstrained optimization within a parallel environment - several of these occuring before parallel computers became widely available. See, for example, Dixon[11], Lootsma[31], Schnabel[41] and their references. Parallel constrained optimization is practically unexplored; there have been few experiences with large scale optimization on parallel computers (for an exception see Lasdon[29]). This paper examines some of the general aspects of vector computing with emphasis on its impact on the field of nonlinear network programming.

Network modeling is a fertile area of mathematical programming. A wide range of real world problems can be formulated as nonlinear networks models. Such problems include stochastic networks[6], hydroelectric power systems control[39], steady-state equilibrium[16,7], balancing of Social Accounting Matrices (SAM)[14], and stock portfolio selection problems[33]. Lasdon and Warren[28] review some of these areas and provide a bibliography. In the past, progressively larger problems were solved by capitalizing on the underlying structure of the network problems[3,34,10,19]. Nevertheless applications still exist - like the U.S. air-traffic control model[35] - that result in problems whose size exceed the capabilities of conventional computer systems. Supercomputers can solve a whole new range of problems that are considered by today's standards to be ultra-large.

Several questions arise with respect to the use of supercomputers. How do we specialize different algorithms to take advantage of the architecture of vector computers, and how do these algorithms compare on such machines? Do our experiences with nonlinear network algorithms on sequential machines still hold true in a vector environment? Does the relative advantage of network algorithms over general purpose codes carry over onto vector machines or not? These questions will be addressed in this paper.

The remaining of the paper is organized as follows: Section 2 provides an overview of current trends in computer design, examining those features of the supercomputer architecture

that are important to the applications programmer. Section 3 describes a general framework for the vectorization of optimization software; these guidelines are applied in vectorizing two non-linear network codes as described in Section 4. Conclusions and computational comparisons with a general purpose code (MINOS) in a vector environment (CRAY X-MP/24 computer) are the subject of Section 5, together with comparative testing among different computer systems. The vectorization framework, the streamlining of the network algorithms, and the computational experiments on the CRAY X-MP/24 are the main contributions of this study.

2. Computer Architecture

Consider a network problem in three spatial coordinates over multiple time periods (e.g. the U.S. air-traffic control model[35]). To achieve an increase in resolution by an order of magnitude would require an increase in speed by a factor of 10**4. The trend in cycle times for computer systems over the last 20 years (Figure 1[4]) suggests that the performance of conventional machines will improve by at most a factor of ten over the next decade. At the same time the performance of parallel computers is expected to improve by a factor of 100-200[44].

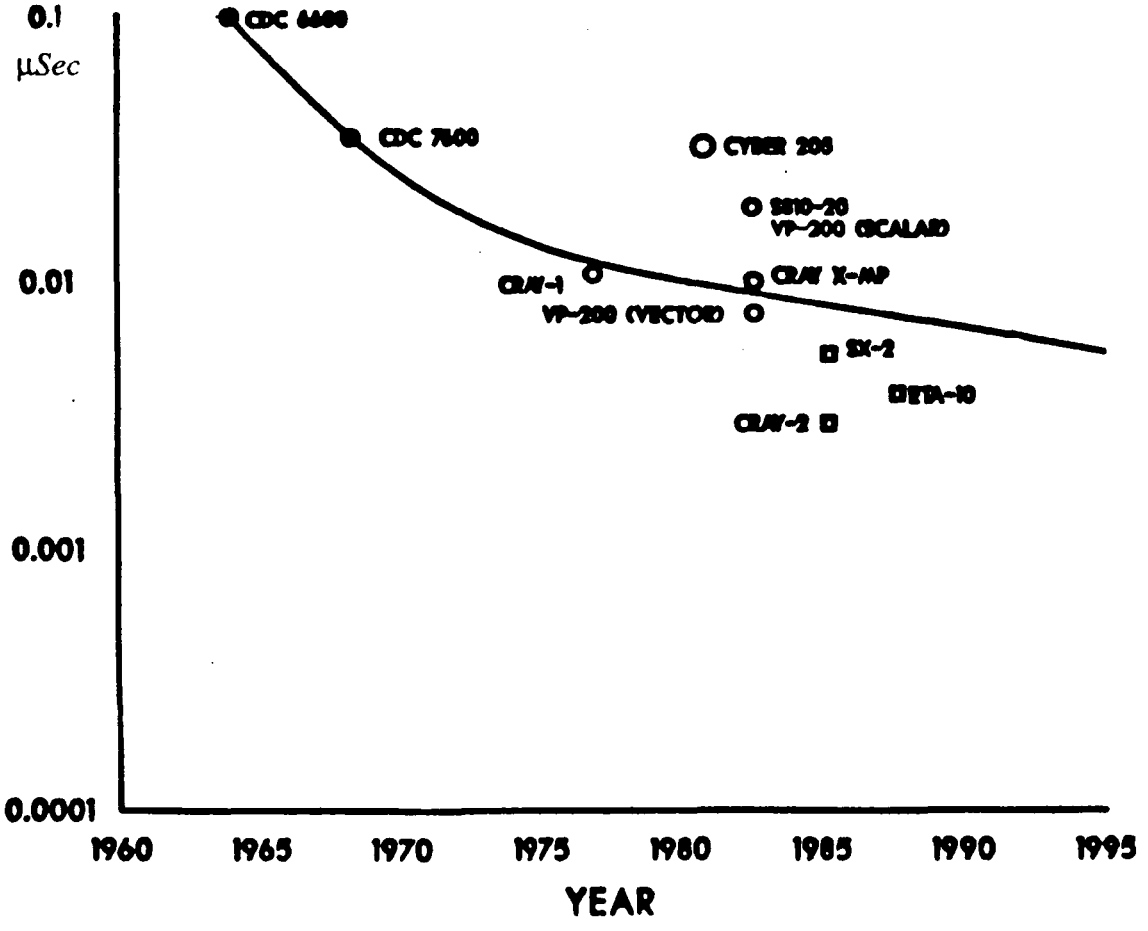


Figure 1: Trend in Cycle Time of Computer Systems †

† Cycle Time $\approx \frac{1}{5.4\,t + 10}$, where t = YEAR - 1964

This performance will be made possible only through the use of innovative computer architectures. The specifics of the computer design become therefore of primary importance for those users who want to exploit the full potential of supercomputers.

In this section we discuss several issues in super-computer design. For more details refer to the papers assembled by Kowalik[26], the work by Warlton[44], Neves[38] or Hockney and Jeeshope[20] and the recently published tutorial on "Supercomputers: Design and Applications"[22]. We describe features of the CRAY X-MP/24 that are important from a programmer's point of view. The choice of the CRAY X-MP as a reference machine is based on the following considerations:

(I) CRAY computers achieve a reasonable balance between general purpose, uniprocessor (but slow) systems, and fast, parallel (but special purpose) machines. They have been shown to achieve relatively high performance on a wide variety of benchmark problems[40].

(II) CRAY computers are the most accessible to the scientific and especially the academic community: 75% of the world's supercomputers are CRAY systems, four out of the six institutions providing supercomputing services to the Academic community in the U.S. have CRAY computers, and two out of the four planned Supercomputer Centers will be equipped with CRAYs[2]. Also twenty-six out of the forty supercomputers installed in six western European countries (as of September 1984) were CRAY machines[15].

Although use of the CRAY seemed more appropriate at this point, it should be pointed out that several other parallel systems are commercially available. It is by no means clear which system is more suitable for a particular application. New designs are constantly appearing either as research projects or on a commercial basis.

2.1. A Trend Analysis

It is beyond the scope of our study to provide a comprehensive analysis of the trends in computer architecture. This is in fact one of the complications arising from parallel computing: there exists a wide variety of architectures each one with its own distinct features that present the programmer with different advantages or disadvantages. A broad analysis indicates that the situa-

tion in designing parallel computers is indeed chaotic.

Parallel computing can be visualized as a domain with two dimensions[5] (see Figure 2). On the one side we have multiple processors operating independently on the same problem, possibly executing different jobs associated with the same problem (*multi-programming*), executing different tasks of the same algorithm (*multitasking*) or executing different portions of the same task (*microtasking*); this is high level parallelism. The second degree of parallelism lies at the level of vector/matrix operations of an algorithm. This is low level parallelism, conveniently called *vectorization*.

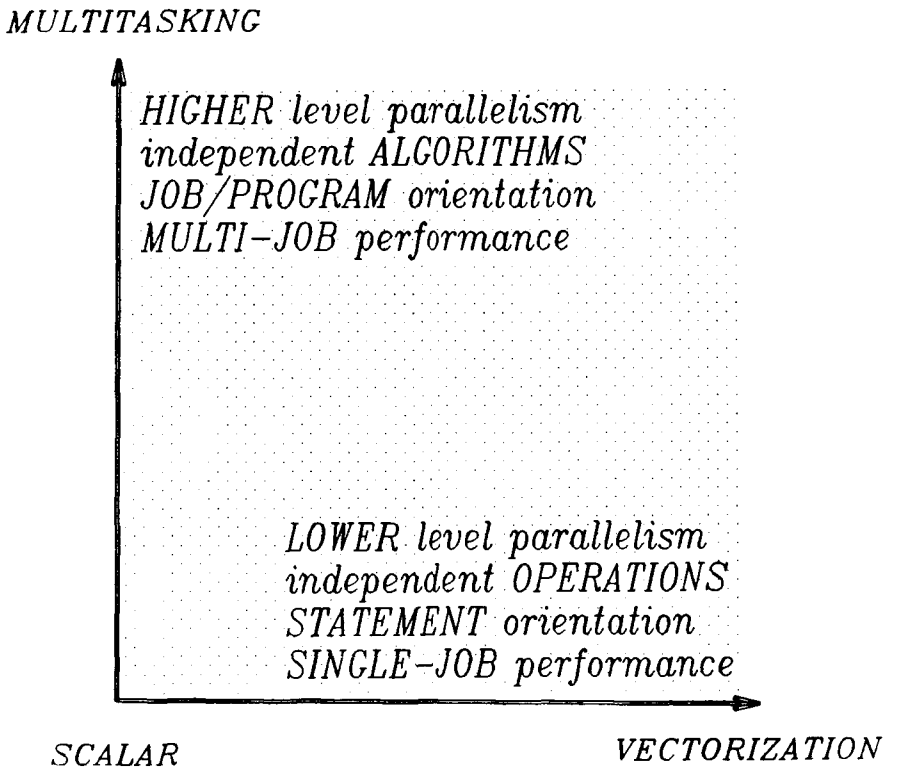


Figure 2: The Two Dimensions of Parallelism

The first dimension of parallelism has been realized in a number of existing computer systems, that in Flynn's taxonomy[17] may be classified as *Multiple Instructions Multiple Data* (MIMD) machines. (Examples: iPSC by INTEL[42], HEP by DENELCOR[43], NYU Ultracomputer[23]).

A further classification of MIMD systems is based on the number of processors available on the system:

Small scale distributed systems < 16 processors

Medium scale distributed systems < 128 processors

Large scale distributed systems <1024 processors

Massively distributed systems >1024 processors

Lastly, multiple processors may be *tightly coupled* communicating with each other through shared registers, shared memory, or by exchanging messages. Or they may be *loosely coupled* through, for example, a local area network incurring high communication costs. See the paper by Cotton[8] for a survey of networking technologies for distributed systems.

The second category of parallel machines (vector processors) fall into Flynn's *Single Instruction Multiple Data* (SIMD) category. (Examples: CRAY X-MP or CRAY-2 by CRAY Research[5,27], CYBER 205 or CYBERPLUS by Control Data[25], FACOM VP100 or VP200 by Fujitsu[32]). These machines are characterized by parallelism at the vector/matrix operations level. In general, they have a number of dedicated functional units (for example, one for addition and one for multiplication). High performance is achieved by executing, in a pipeline fashion, operations on long vectors or matrices. Vectors may be fetched directly from memory (e.g. CYBER 205), or may be loaded through fixed length registers (CRAY systems), or through reconfigurable registers (FACOM). Some machines, like the CRAY, may address vectors in non-contiguous storage, while on other machines, like the CYBER 205, this is not possible. As a further complication, vector computers are designed with a number of processors that may operate independently.(For example CRAY X-MP comes in versions with two or four processors).

This multitude of computer designs provides the programmer with new challenges in the design of optimization algorithms. It is unlikely that computer architecture will stabilize to a particular configuration in the forseeable future, and a universal solution to the problem of parallel software development is unlikely to be found. Lootsma[31] identifies some of the issues arising from the evaluation of algorithms in parallel environment, and Mulvey and Zenios [36] address the problem of managing large software systems in the face of rapidly changing architectures.

We describe in Section 3 a general framework for streamlining optimization algorithms for a vector environment. Our attention is limited to a well specified architecture - that of a CRAY X-MP computer - which is now briefly examined. (Our framework, however, may be extended to the case of other machines with similar organization.)

2.2. CRAY X-MP Architecture for Application Programmers

The CRAY X-MP is a vector, register-to-register computer. This means that vectors are loaded from memory into registers before being processed; results are stored in registers before returning to memory.

Figure 3 is a schematic representation of the CPU architecture of a CRAY X-MP; for a more detailed description refer to[5, 27, 1]. The features of the architecture that are of interest to the mathematical programmer are summarized below :

(1) Dedicated functional units for **floating point vector** ADDITION, MULTIPLICATION, and RECIPROCAL APPROXIMATION.

(2) Dedicated functional units for **integer vector** ADDITION, LOGICAL, and SHIFT operations.

(3) Dedicated functional units for **integer scalar** ADDITION, LOGICAL, and SHIFT operations.

(4) Vector functional units operate in a **pipeline** fashion: the first result appears after a start-up period, and remaining results appear with the rate of one every clock period.

(5) Eight **vector** and eight **scalar** registers. Vector registers are 64 real words long each.

(6) Four parallel paths to/from memory: two for vector LOAD, one for vector STORE, and one for independent I/O.

(7) Four million words of memory, (32 Mbytes), organized in 32 interleaved memory banks.

(8) Large Solid-state Storage Device (SSD) with size 32 million word (128 Mbytes) and high block transfer rate. (Data transfer rate ~ 250 times faster than disk at 1000Mbytes/sec ; SSD access times ~ 100 times faster than disk at 0.5ms)

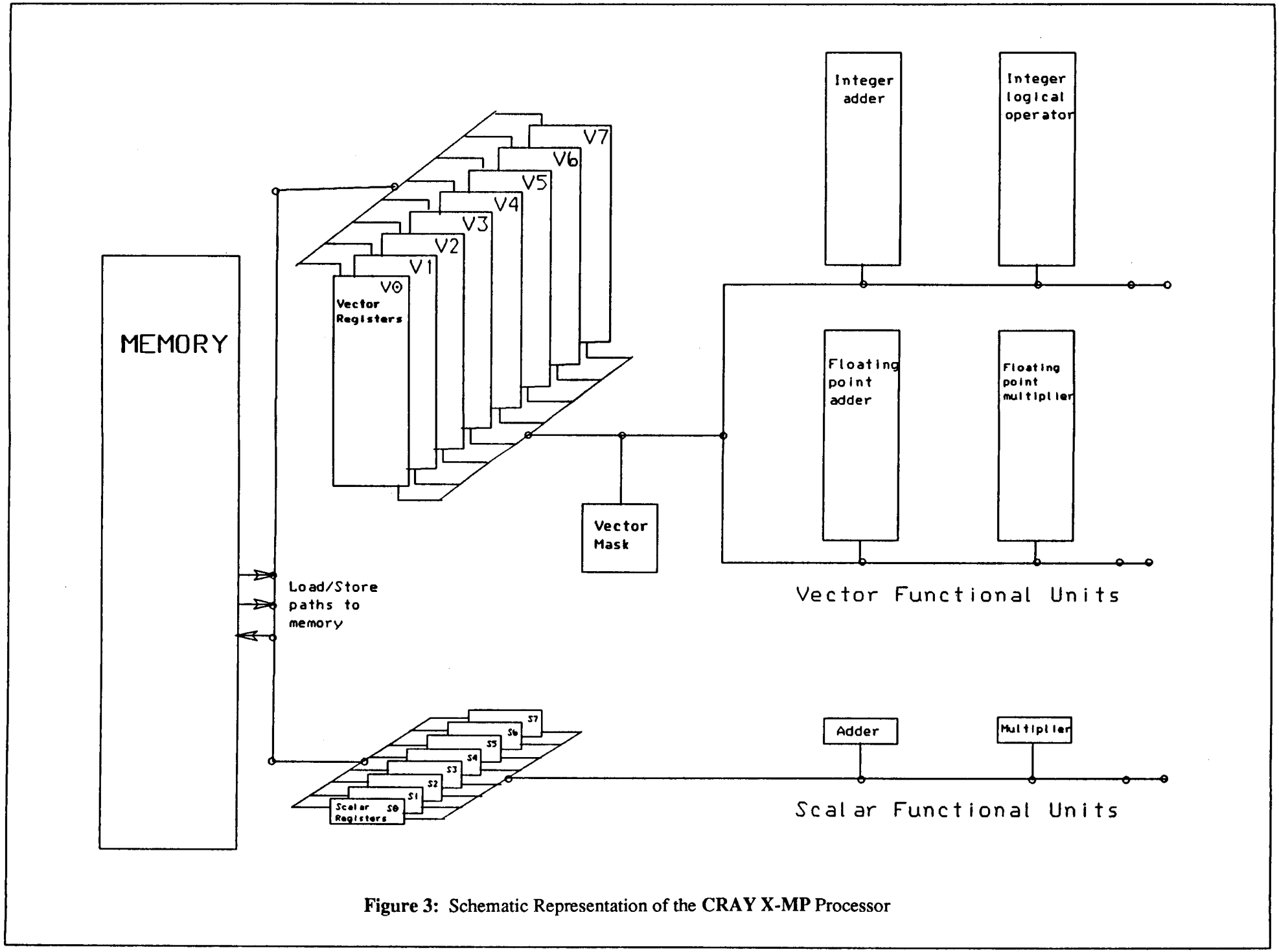


Figure 3: Schematic Representation of the **CRAY X-MP** Processor

We proceed to describe how these special features are utilized efficiently by a Fortran program. The CRAY Fortran Compiler (CFT) will generate machine instructions for the usage of the vector functional units and registers. The compiler, however, will not generate vector code in many instances. It is therefore the programmer's responsibility to present the compiler with constructs that will vectorize. (This view is reinforced by the computational results of Section 4 and in particular Table 2).

2.2.1. Vectorization of DO Loops

The primary vehicle of vectorization is the iterative procedure (i.e. inner DO loops). If a loop is executed N times, where N = 64*K + r , the compiler will generate instructions for a (short) loop of length r, and then will process the additional K segments - recall that 64 is the size, in words, of the vector registers. This strategy is possible only when operations are identical - i.e., no branching statements are encountered during execution of the loop. Also data in the processed vectors have to be independent - i.e., no vector element should depend on the value of some other elements computed during execution of the loop.

The vectors are processed by the independent functional units in a pipeline fashion: the first result will appear from the functional unit after a start-up period, conveniently called the *f-unit*. Subsequent results appear at the rate of one every cycle. In this environment we would rather process large vectors so that the overhead of f-units is diminished.

2.2.2. Functional Unit Parallelism

The presence of multiple functional units may be exploited in two ways:

(1) Functional Unit Overlap: functional units may operate independently of each other. For example, while one unit is adding two vectors the multiplication unit may process two different vectors. The existence of multiple paths to/from memory make it possible to load the vectors for the multiplication while storing the results of addition.

(2) Functional Unit Chaining: the results of one functional unit may be fed directly into a second unit. See, for example, Figure 4 which describes chaining of multiplication with

addition during execution of the $(a \cdot \bar{x} + \bar{y})$ operation, called SAXPY using the BLAS notation[30] (scalar a times vector $\bar{x}$ plus vector $\bar{y}$).

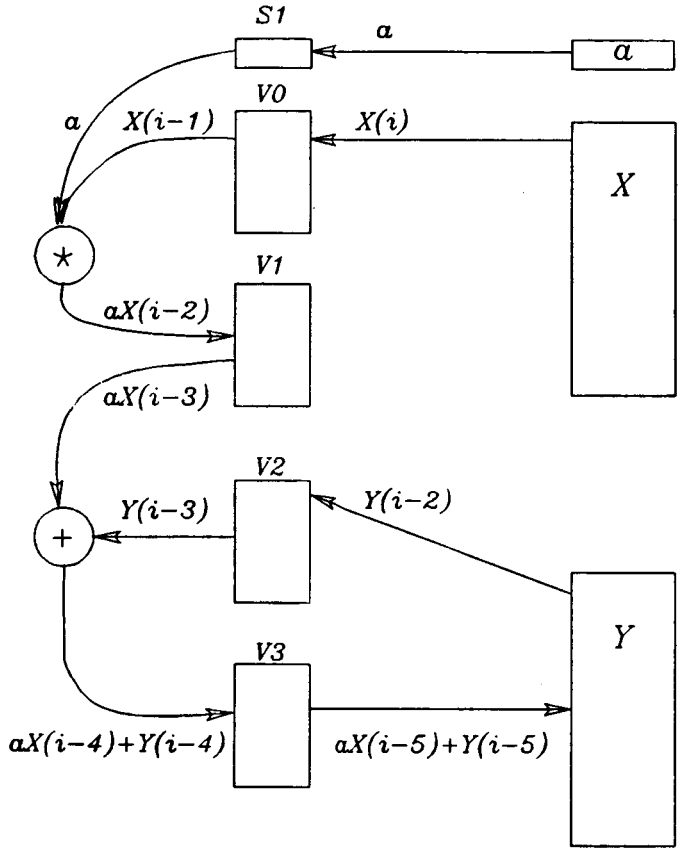


Figure 4: Functional Unit Chaining in the SAXPY Operation

2.2.3. Sparse Vector Operations

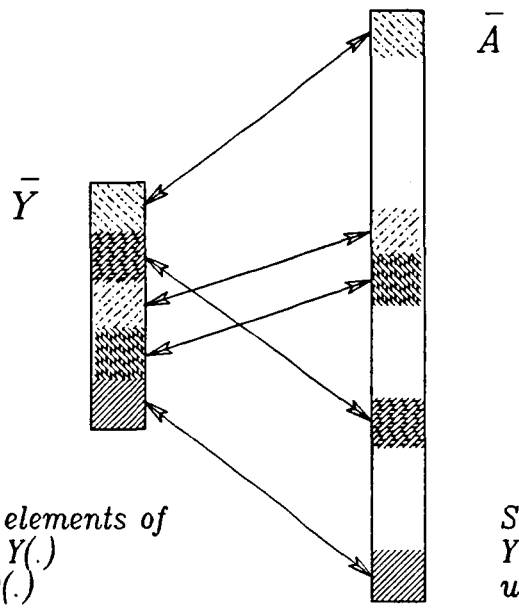
In most large scale optimization systems some kind of sparsity is present, and is exploited by the software. This in general requires at least one level of indirect addressing. Indirect and nonlinear addressing will inhibit vectorization. Hence we use a three step procedure for sparse operations:

(1) GATHER the non-zero components of the sparse vector, according to the indirect addressing list.

(2) Process the gathered data in vector/pipeline mode

(3) SCATTER the results using the indirect addressing list, to their corresponding locations.

This technique transforms a non-vectorizable code into a part that will vectorize (2), and two parts (1,3) for which most supercomputers have readily available routines coded efficiently at the assembly language level. (GATHER and SCATTER are implemented at the hardware level on the CRAY X-MP and are basic instructions for the CYBER 205). See Figure 5 for an example; refer to the paper by Dembart[9] for an extensive discussion of sparse loop operations that may be vectorized using SCATTER/GATHER operations. Operations (1), (2) and (3) exploit functional unit parallelism (chaining and overlap). The efficiency with which they are performed counterbalances the involved overhead, especially in processing long vectors.



GATHER the non-zero elements of vector A(.) into vector Y(.) using index array IND(.)

```
DO for all  I = 1 , N
   Y (I) = A ( IND(I) )
ENDDO
```

SCATTER the elements of vector Y(.) into the sparse vector A(.) using index array IND(.)

```
DO for all  I = 1 , N
   A ( IND(I) ) = Y (I)
ENDDO
```

Figure 5: The **SCATTER/GATHER** Operations for Sparse Vectors

2.2.4. Memory Access

The CRAY X-MP/24 has four million words of real memory, organized in 32 interleaved memory banks. The reason why memory is interleaved in banks is (fortunately) not of interest to the applications programmer - refer to[1]. The fact however that exactly 32 banks are available is of importance: it is possible to issue instructions to transfer words into/out of memory at the rate of one word per clock period (cp). However there is a period - called *bank cycle* - during which the load/store is completed, and during this period the memory bank involved is *busy*. The memory bank cycle for the X-MP is 4cp, and a *memory bank conflict* will occur if two load/store instructions refer to the same bank within 4cp. How can this happen? Since arrays in Fortran are stored as a long contiguous vector, over 32 banks, every 33rd element of a vector will be in the same bank. Hence the program should avoid referencing a vector by a stride of 32, and avoid simultaneous operations on vectors whose location in memory is offset by 32 words. Performance will degrade by a factor of four in the above mentioned cases. Degredation by a factor of two will occur in addressing elements of vectors offset by 16 words, since a memory bank conflict will occur every second instruction.

3. A Vectorization Framework

In streamlining a software system for a parallel environment we consider vectorization at three levels:

(1) Local Software vectorization

(2) Global Software Vectorization

(3) Global/Local algorithm vectorization.

At the first level, the program is examined at the subroutine level, and redesigned to take advantage of the computer features. At the second level the whole implementation of the algorithm is examined; data structure and application choices are redesigned to take advantage of the machine. At the third level the choice and the design of the mathematical algorithm is reconsidered, in the light of machine architecture. The later level of vectorization is a challenging area of research which is practically unexplored: current algorithms for nonlinear network programming are the result of years of research and experience from different researchers and we are restricted, at least temporarily, to consider vectorization of existing algorithms. Machine architecture, however, should be considered as an equally important factor with the problem structure/characteristics in the design of new algorithms. This philosophy is reflected in the innovative algorithm for linear programming by Karmarkar[24]. The fact that the algorithm is suited for parallel processing is often cited as one of its advantages over the simplex method.

Local software vectorization is the first issue to be considered. While perpetuating inefficiencies of past designs, code performance may improve by as much as an order of magnitude and is in general easier to pursue. Global software vectorization should be considered when it is recognized that current design (data structure, etc.) totally inhibit vectorization at the local level. It is in general more expensive in terms of computational and human resources, but together with local vectorization it may result in substantial improvement in the performance of the algorithm.

We discuss below the steps that ought to be taken in vectorizing optimization software.

STEP-1: Use a time-analysis of the algorithm to determine the computationally intensive parts of the program. Over 80% of the time of most application programs is spent in less than 20% of the code. Vectorization efforts should be concentrated on this part of the code.

STEP-2: Use, where possible, computational kernels that are currently available on most vector computers, and are fine-tuned for the particular architecture.

STEP-3: Rearrange/modify the order of operations to make possible the use of computational kernels.

STEP-4: Rearrange the order of nested DO loops. Since only inner DO loops will vectorize, make those loops over the biggest index.

STEP-5: Operate on the biggest dimension of two dimensional arrays.

STEP-6: Unroll DO loops. This technique was originally proposed by Dongarra and Hind[12]; instead of executing a loop over I=1 to N with stride 1, use stride K and write explicit code to account for the calculations over I,I+1,...I+K-1. This technique eliminates the overhead of checking for termination conditions (I> N) and also enforces chaining and/or functional unit overlap. The depth at which unrolling is performed, i.e., the value of K, depends on the particular operations to be executed.

STEP-7: Within DO loops avoid statements that inhibit vectorization:

- IF statements
- GOTO statements
- CALL statements
- Nonlinear vector references
- Vector dependencies

STEP-8: If Steps 2 to 7 were successful some other part of the algorithm is now the most computationally intensive, and has to be reexamined; go to STEP-1 and repeat. If the most computationally intensive part of the algorithm is vectorized, no further

improvement can be achieved and the process may terminate. If no substantial improvement in the performance of the algorithm was achieved in executing steps 1 to 8 proceed to STEP-9.

STEP-9: Reconsider global aspects of the algorithm that may inhibit vectorization - typically the data structure. Are there any changes possible that will make Steps 1 through 8 effective, and if yes do the anticipated improvements justify the effort? If the answer is yes go to STEP-1 and repeat. Otherwise we are dealing with an inherently scalar algorithm, and the choice of algorithm or problem formulation will have to be reconsidered.

Conclusion : Vectorization of a program is an iterative process involving both knowledge of the algorithm and understanding of the architectural features of the computer. Although attempts are being made to design efficient vectorizing compilers, and array/vector extensions to the ANSI Fortran are discussed (See Chapter 4 of "Supercomputers and Applications"[22]), the burden is currently on the applications programmer for the efficient vectorization of his software.

In the next section we use this framework in the context of two nonlinear network codes: in the one case with very encouraging results in terms of improved performance, and in the other case with less desirable, yet illuminating conclusions.

4. Case Studies in Nonlinear Network Programming Vectorization

In this section we describe how the vectorization framework developed earlier was applied in the context of two codes for nonlinear network programming. A nonlinear generalized network problem is defined in the form :

[NLGN] Minimize $F(\bar{x})$

Subject to :

$$A \cdot \bar{x} = \bar{b}$$

$$\bar{l} \leq \bar{x} \leq \bar{u}$$

where :

$F(\bar{x})$: convex objective function

$\bar{x}$: vector of decision variables

A : generalized network constraint matrix with two non-zero entries in every column

$\bar{b}$: vector of supplies and demands

$\bar{l}/\bar{u}$: vectors of lower/upper bounds

NA : number of arcs in the network

NN : number of nodes in the network

The first code (NLPNETG) is based on the primal truncated Newton algorithm, implemented using an active set strategy whereby the constrained matrix is partitioned into a basic, superbasic and non-basic part: $A = [B \mid S \mid N]$. For a description of the underlying algorithm and discussion of implementation issues refer to Ahlfeld et al[3]. The second code (GNSD) is based on the simplicial decomposition algorithm of Hohenbalken[21], with inexact solutions to the master problem. Refer to Mulvey et al. [34] for a description of the algorithm and its implementation. The reader who is unfamiliar with developments in computational nonlinear programming will find the book by Gill et al. [18] useful.

The computational testing was carried out on a CRAY X-MP/24 available at Boeing Computer Services (BCS). This CRAY model has two independent CPUs and comes with 4MWords of real memory - only one CPU was used by our programs due to operating restrictions by BCS. The CRAY FORTRAN compiler CFT1.13 was used in all cases. The test problems were

collected from three areas of application and one set of randomly generated problems. See Table 1 for a description of their characteristics.

PROBLEM	Size (Nodes/Arcs)	Free arcs at optimum	Condition No. of Reduced Hessian	Objective value	l_∞ Norm of Reduced Gradient	Description
PTN30	30/ 46	15	$\geq 10^4$	-.3239322E5	.0045	Dallas Water Distribution models
PTN150	150/ 196	44	$\geq 10^4$	-.4819730E5	.0020	
PTN660	666/ 906	240	$\geq 10^6$	-.2061074E6	.0160	
SMBANK	64/ 117	54	$\geq 10^4$	-.7129290E7	.0003	Matrix balancing models
BIGBANK	1116/2230	946	$\geq 10^8$	-.4205693E7	.0004	
GROUP1ac	200/ 500	100	$\geq 10^3$	.1011792E5	.0060	Randomly generated, strictly convex networks
GROUP1ad	400/1000	119	$\geq 10^4$	.3834884E5	.0049	
MARK3	857/1710	4	$\geq 10^2$	-.1145214E6	.0020	Markowitz portfolio construction

Table 1 : Test Problems

4.1. Truncated Newton Algorithm

Once the software system NLPNETG was installed on the CRAY it became evident that the vectorizing compiler had only marginal effect on performance (average improvment in solution time ~ 14%) - refer to the first two columns of Table 2. This behavior was anticipated, since NLPNETG makes extensive use of sparse matrix handling techniques, and employs data structure that are inherently scalar. A timing analysis of the program for a typical test problem (PTN660) reveals that the following parts of the program are the most computationally demanding:

1. Conjugate Gradient routines (CG)
2. Function/Gradient/Hessian evaluation (CALFGH)
3. Active set generation routine (PVCOL)
4. Basis handling routines (REVMAX)
5. Active set storage routine (STORE)

Problem	NLPNETG Solution times (sec)		
	Without vectorization	Compiler vectorization	User vectorization
PTN150	0.328	0.317	0.165
PTN660	2.435	2.316	1.402
SMBANK	0.358	0.327	0.177
BIGBANK	244.107	225.900	58.896
STICK4	*	*	2.925
GROUP1ac	18.668	17.215	4.983
GROUP1ae	*	*	49.454
MARK3	*	*	2.131
Av. ratio	1.00	0.86	0.22

* Problem not solved with this option

Table 2 : Vectorization of **NLPNETG** on the **CRAY X-MP/24**

6. General purpose sorting routine (SORT)

Point A of Figure 6 indicates the percentage of CPU time spent in every one of these routines, and the total time for solving this problem. The conjugate gradient (CG) routines are the most computationally demanding for this code.

The CG routines implement the linear conjugate gradient algorithm for solving Newton's equations for the search direction on the active set:

$$(\bar{Z}'\cdot H\cdot\bar{Z})\cdot\bar{p}_s = -\bar{Z}'\cdot\bar{g}$$

where :

$\bar{p}_s$, *descent direction on the active set*

$\bar{g} = \nabla F(\bar{x})$, *gradient vector* $(NA-long)$

$H = \nabla^2 F(\bar{x})$, *Hessian matrix* $(NA \times NA)$

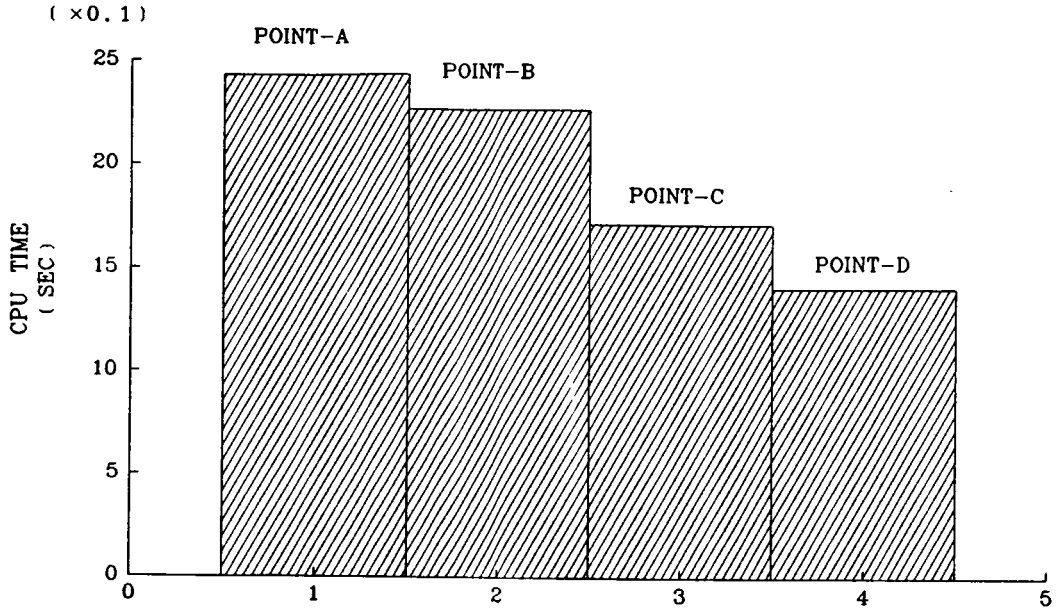
B , *square nonsingular basis matrix* $(NN \times NN)$

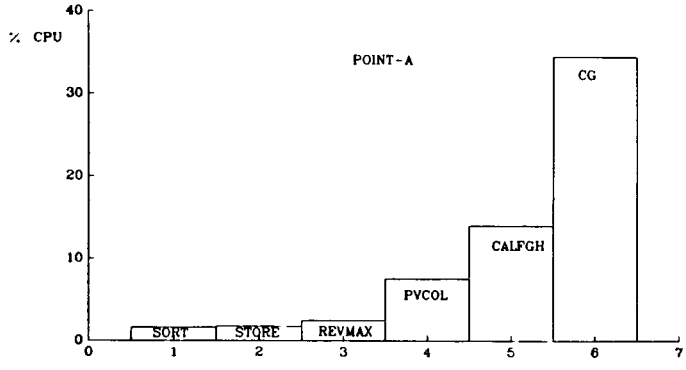
S , *rectangular matrix of active set columns* $(NN \times NS)$

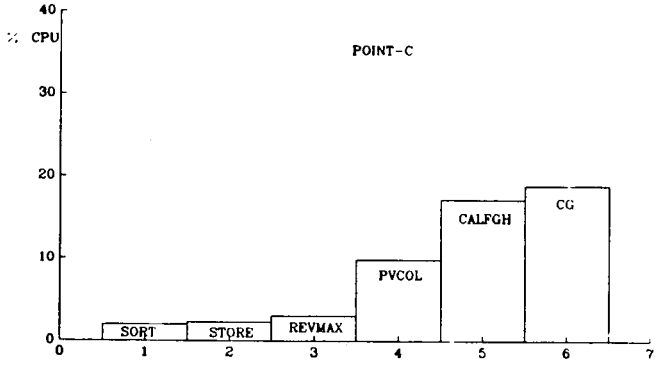
NS is the number of superbasic variables in the current subspace

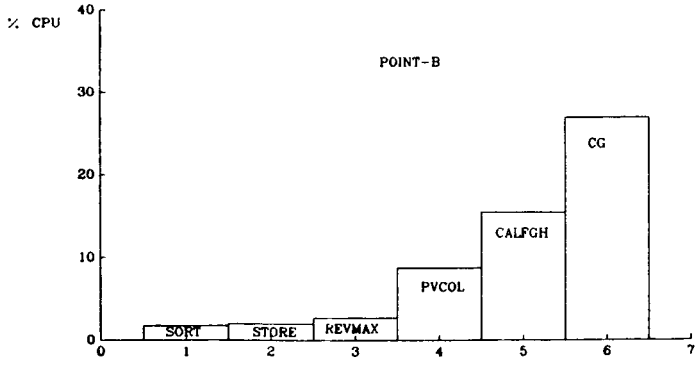
$$Z = \begin{bmatrix} -B^{-1}\cdot S \\ I \\ 0 \end{bmatrix}, \text{ projection matrix } (NA \times NS)$$

For a detailed description of the conjugate gradient algorithm refer to Gill et al[18]. The advantage of this approach for large scale computing, however, is that it does not require explicit









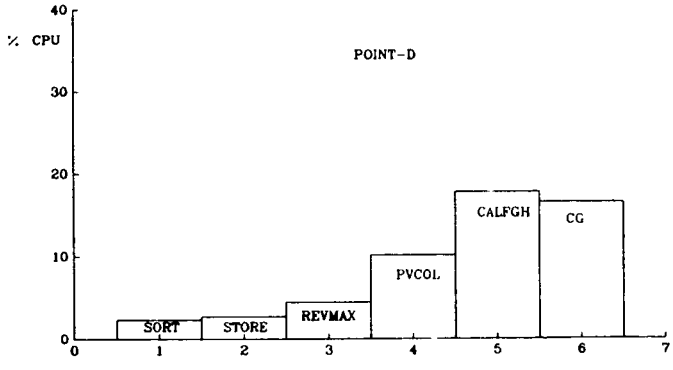


Figure 6: Vectorization of **NLPNETG**

computation of the reduced Hessian ($Z^{\prime} \cdot H \cdot Z$); instead it forms a series of products of the form $Z \cdot \bar{v}$, $H \cdot \bar{v}$ and $Z^{\prime} \cdot \bar{v}$, where $\bar{v}$ is a dense vector defined iteratively during execution of CG. We next describe how the three matrix-vector products in CG were coded for maximum efficiency on the CRAY X-MP/24.

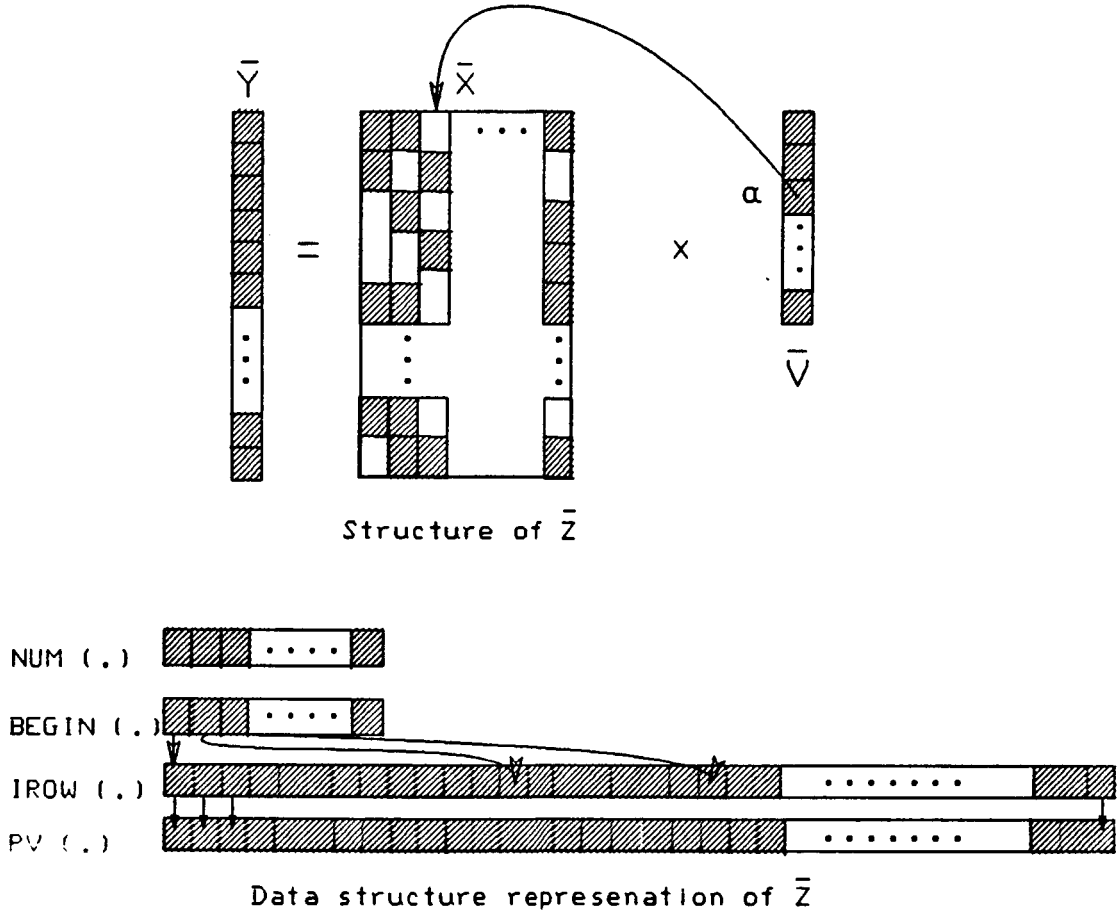


Figure 7 : Vectorization of the Product $Z \cdot \bar{v}$

4.1.1. Vectorization of Z·v̄

The matrix Z is partitioned into three parts: an identity matrix I for the superbasic variables, a null matrix 0 for nonbasic variables and a sparse matrix $(-B^{-1} \cdot S)$ for basic variables. The products of $\bar{v}$ with I and 0 are computed in the obvious way. The product of $(-B^{-1} \cdot S) \cdot \bar{v}$ deserves special attention. For convenience we refer to Z as being the matrix $(-B^{-1} \cdot S)$.

We store Z columnwise, as a sparse matrix in vector $PV(\cdot)$. We use a $BEGIN(\cdot)$ pointer array to indicate the starting address of every column in $PV(\cdot)$. A $NUM(\cdot)$ array indicates the number of

non-zero entries in the column, and a pointer $IROW(\cdot)$ indicates the corresponding row number of every non-zero element - See Figure 7.

The obvious way to compute $\bar{Y} = Z \cdot \bar{v}$ on a scalar computer is the following:

```
Step X  DO for all I = 1, NA
          Y (I)= 0.0
        ENDDO

Step Y  DO for all J = 1, NS
          JBEG = BEGIN (J)
          JEND = JBEG + NUM (J) - 1
          VJ   = v (J)
          DO for all K = JBEG, JEND
             Y (IROW (K)) = Y (IROW (K)) + VJ * PV (K)
          ENDDO
        ENDDO
```

On a vector machine some modifications are necessary - though for this example the operations performed do not change. The first DO loop (Step X) is unrolled to a depth of four, and efficient vector instructions are generated by the compiler:

```
Step XV DO for all I = 1, NA, 4
          Y (I  )= 0.0
          Y (I+1)= 0.0
          Y (I+2)= 0.0
          Y (I+3)= 0.0
        ENDDO
```

The inner DO loop of Step Y is a sparse SAXPY operation[30]. It is treated in the following fashion:

```
Step YV DO for all J = 1, NS
          JBEG = BEGIN (J)
          JEND = JBEG + NUM (J) - 1
          VJ   = v (J)
          DO for all K = JBEG, JEND
             Y´ (K)= VJ * pv (K)
          ENDDO
          SCATTER  Y´ (IROW (K)) → Y´´
          DO for all I = 1, NA
             Y  (I)= Y (I) + Y´ (I)
          ENDDO
        ENDDO
```

The SAXPY operation is a part of BCS/VectorPack library, coded in CRAY assembly language (CAL) for maximum efficiency, so the vectorized code is actually written as:

```
Step YV DO for all J = 1, NS
        JBEG = BEGIN (J)
        NUME = NUM (J)
        VJ   = v (J)
        call SAXPY (Y, IROW(JBEG), VJ, PV(JBEG), NUME)
        ENDDO
    ENDDO
```

The improvement in the efficiency of NLPNETG following these modifications is depicted by Point B of Figure 6.

4.1.2. Vectorization of $H \cdot \bar{v}$

The Hessian matrix H is symmetric, and is stored using a modification of the Yale sparse matrix storage scheme. The nonzero entries of the matrix are stored row-wise in vector $H(\cdot)$. A pointer array $IH(\cdot)$ points to the starting address of every row in $H(\cdot)$, and pointer $JH(\cdot)$ indicates the corresponding column of all nonzero elements. Diagonal elements are stored even if the numerical value is zero. The obvious way to compute $\bar{Y} = H \cdot \bar{v}$ on a scalar machine is the following:

```
Step X  DO for all I = 1, NA
        Y (I) = 0.0
    ENDDO

Step Y  DO for all I = 1, NA
        SUM  = 0.0
        IBEG = IH (I)
        IEND = IH (I+1) - 1
        DO for all J = IBEG, IEND
           POINT = JH (J)
           SUM   = SUM + v(POINT) * H(J)
           Y(POINT) = Y(POINT) + v(I) * H(J)
        ENDDO
        Y(I) = Y(I) + SUM
    ENDDO
```

Although Step X will vectorize efficiently on the CRAY, the indirect addressing of the inner DO loop in Step Y inhibits vectorization. The vectorization of $H \cdot \bar{v}$ requires the execution of operations in different order than in a scalar code, in order to make use of the BCS/VectorPack routine SAXPY and SDOT - inner product of a sparse vector with a dense vector :

Step WV GATHER Y′ ← H(IH)

Step XV DO for all I = 1, NA, 4
Y (I) = Y′ (I) * v(I)
Y (I+1) = Y′ (I+1) * v(I+1)
Y (I+2) = Y′ (I+2) * v(I+2)
Y (I+3) = Y′ (I+3) * v(I+3)
ENDDO

Step YV DO for all I = 1, NA such that (IH(I+1)-IH(I)-1) not equal 0
NUM = IH(I+1)-IH(I)-1
IPOINT = IH(I) + 1
Y (I) = Y(I) + SDOT (H(IPOINT), IH(IPOINT), NUM, v)
call SAXPY (Y, JH(IPOINT), v(I), NUM)
ENDDO

The improved performance of NLPNETG following these modifications of $H \cdot \bar{v}$ is depicted by POINT D in Figure 6. A few irregularities - as seen by a Fortran programmer on a sequential machine - require some explanation :

- In Step WV the diagonal components of H are GATHERed in a temporary array $\bar{Y}'$, so as to initialize the vector $\bar{Y}$ to the values of diag $(H \cdot \bar{v})$ in an efficient way.

- Off-diagonal components of H , for those rows where such elements do exist, contribute to the product $H \cdot \bar{v}$ twice: once due to their presence in the upper triangular matrix, and once due to the presence of a symmetric element below the diagonal. This contribution is computed twice: once in the SAXPY statement and once in the SDOT statement. While the number of operations performed by this code is <u>not</u> optimum, the calculations are performed in an efficient vector/pipeline mode.

- The DO loop in Step YV is defined over values for which off diagonal elements are present, thus avoiding the overhead for calling the VectorPack routines with zero components. On a scalar machine no significant difference is detected between executing a loop zero times, or having an IF/GOTO statement skip the loop altogether. POINT C (as opposed to POINT D) of Figure 6 indicates the degredation in performance that may occur by following the same programming practice on the CRAY.

In implementing the above changes of NLPNETG care was taken to avoid memory bank

conflicts, merely by dimensioning the arrays by non-multiples of 32. Solving a sample problem (GROUP1ad), with intentional bank conflicts gives the following results:

- Without bank conflicts: solution time 87.5 sec
- With bank conflicts : solution time 102.8 sec

We observe a substantial degredation in performance as a result of inadequate knowledge of the computer characteristics.

4.2. Simplicial Decomposition Algorithm

The second candidate for vectorization was code GNSD, based on the simplicial decomposition algorithm[21,34]. This code challenges the vector machine architecture in two domains:

(I) Performing operations on sparse vector/matrices that are stored in space economizing form.

(II) Performing matrix operations on graph data structures.

While sparse vector operations can be handled using a GATHER/SCATTER approach, much in the same way as with NLPNETG, only marginal improvement was achieved. This behavior was anticipated since almost 90% of the time in GNSD is spent solving the linear generalized network subproblem, with operations performed on graph data structure. This part of the algorithm is inherently scalar, and in the absence of alternative algorithms or data structures we concentrate on vectorizing the master problem. This involves the repeated solution of the Newton equations:

$$(D^{T} \cdot H \cdot D) \cdot \bar{p} = -D^{T} \cdot \nabla F(D \cdot \bar{w})$$

where :

$Y = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_n)$ *is the matrix whose columns are the extreme points (vertices)*

$\bar{w} = (w_1, w_2, \ldots, w_n)$ *are associated weights* for *the vertices*

$D = (\bar{y}_1 - \bar{y}_n, \ldots, \bar{y}_{n-1} - \bar{y}_n)$ *is the derived linear basis representing the active simplex*

Note that D is not computed explicitly, but instead the upper triangular of the symmetric matrix $C = D^{T} \cdot H \cdot D$ is computed in the form:

$$C = Y^{T} \cdot H \cdot Y - Y^{T} \cdot H \cdot Y_n - Y_n^{T} \cdot H \cdot Y + Y_n^{T} \cdot H \cdot Y_n$$

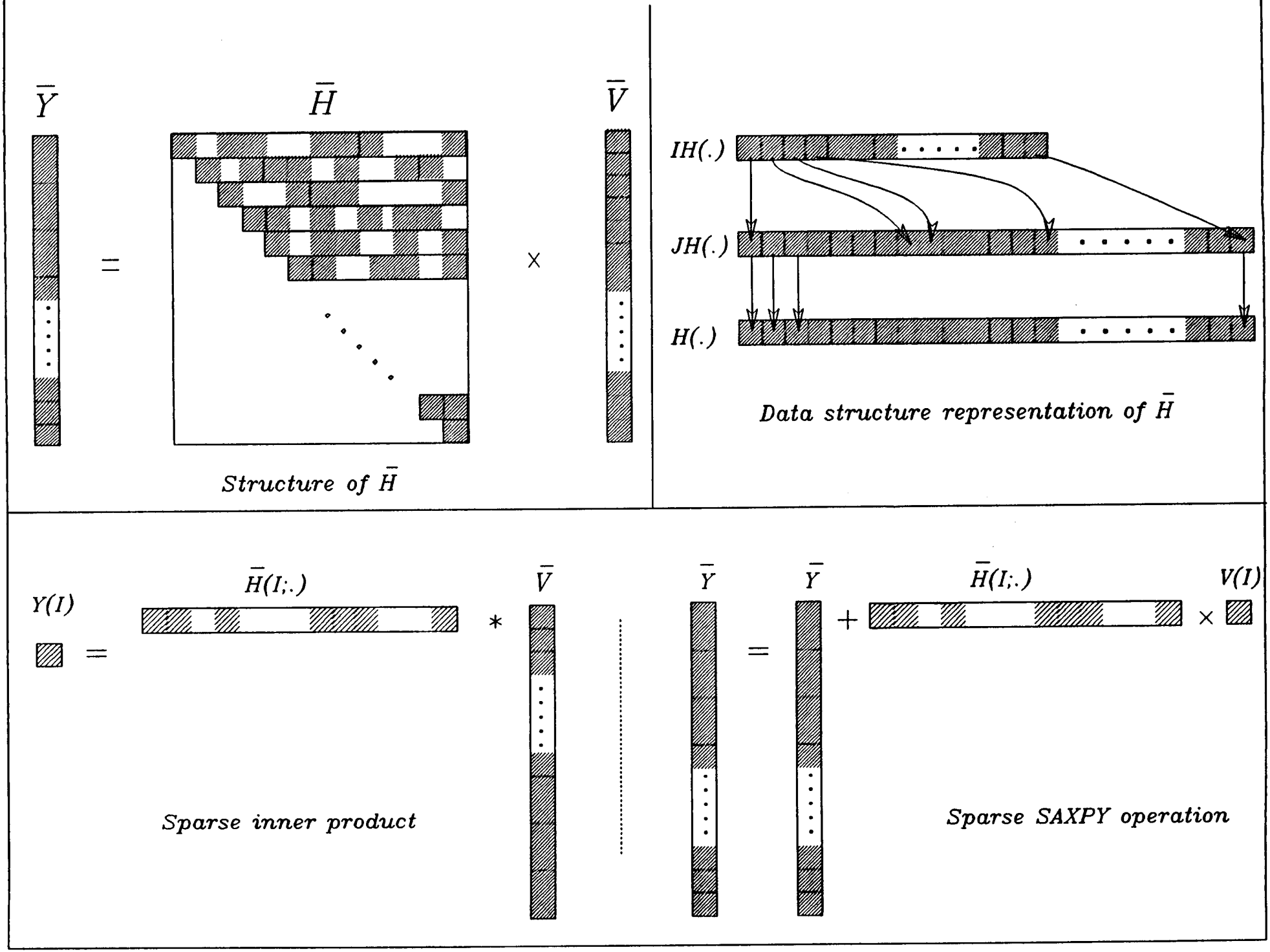


Figure 8 : Vectorization of the Product $\bar{H} \cdot \bar{v}$

where Y_n is a conformable matrix, with its columns equal to $\bar{y}_n$. These products were streamlined for the CRAY X-MP using the same techniques employed in the vectorization of NLPNETG.

Our local software efforts had only marginal improvement (2%) on the compiler vectorized code, which in turn was only marginally faster than the scalar performance (5%) -- see the first three columns of Table 3. Repeated applications of Steps 1-8 of our vectorization framework, indicated that some global aspects of the implementation had to be reconsidered. In particular, it became evident that the data structure for storing the vertices was inhibiting efficient vectorization of the master problem.

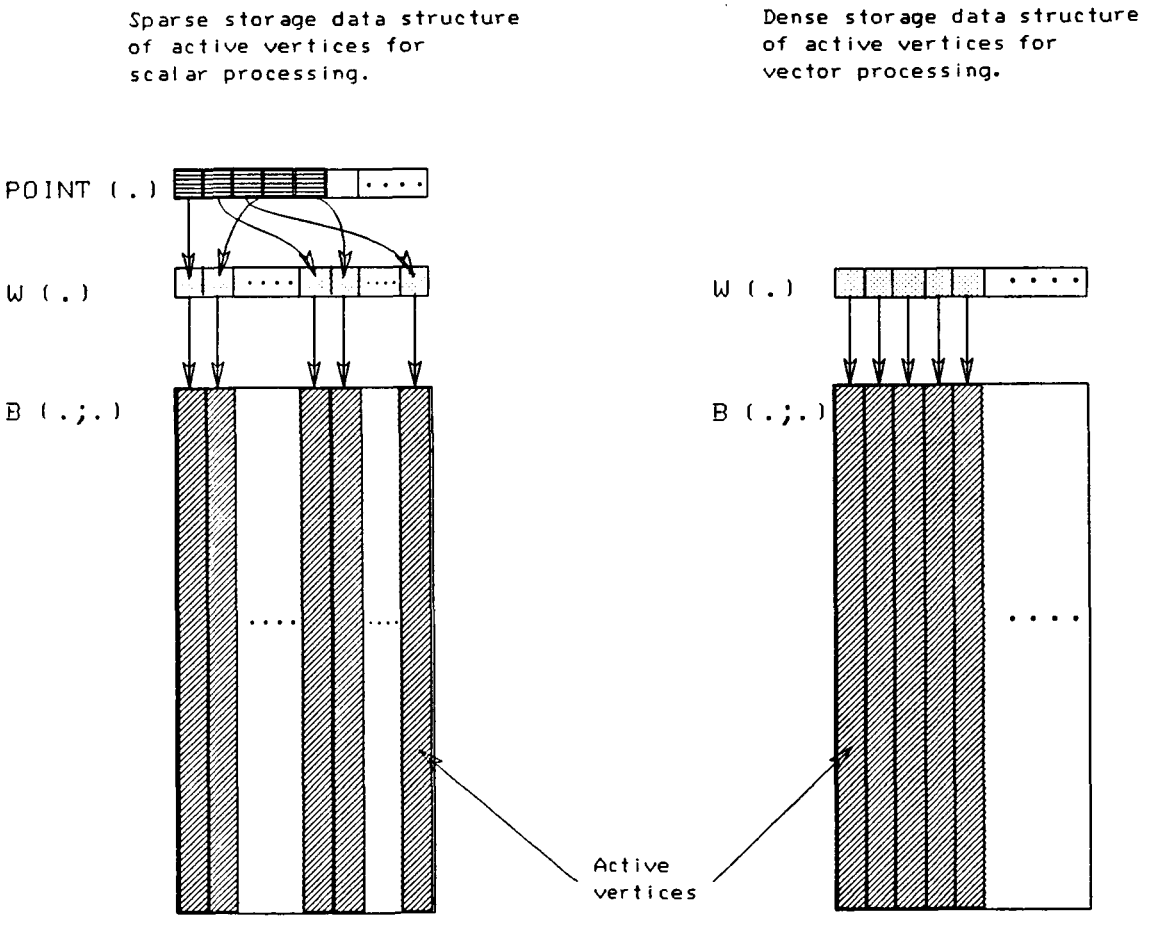


Figure 9: Active Vertices Storage for Simplicial Decomposition

The vertices of the active simplex are represented by NA-long vectors, stored in matrix Y. To avoid rearranging the matrix every time a new vertex was added or an active vertex dropped,

we implemented a pointer scheme, indicating the active vertices in Y and their corresponding weights - see Figure 9. This pointer was used in indirect addressing during computation of C, and an alternative (also simpler) scheme was used, where the active vertices were ordered in the first n columns of Y, and the weights were stored in the same order. This strategy removed the problem of indirect addressing, but additional work is needed to keep the columns of Y in the right order, every time a vertex is dropped. The results of this change in data structure on the CRAY and a VAX 11/750 are shown in Figure 10.

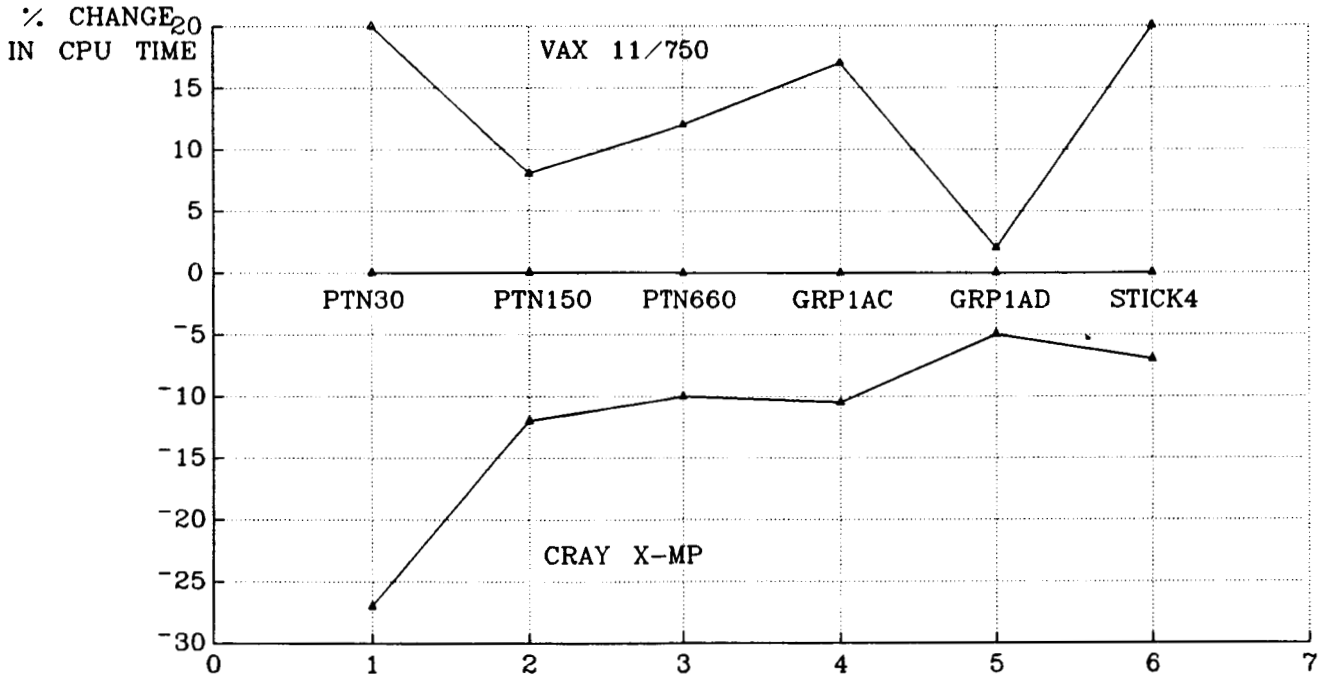


Figure 10: Performance of Vectorized GNSD on a **CRAY X-MP/24** and a **VAX 11/750**

On the CRAY the data rearrangement is executed in vector mode, and the presence of multiple paths to/from memory makes this operation very efficient. Little overhead is introduced in rearranging Y. The resulting code has improved efficiency; as much as 25% for some problems with average improvement 13%. On the sequential machine (VAX), however, little is gained by removing one level of addressing from the master problem procedures. The overhead of rearranging Y is high; the vectorized code is less efficient - as much as 20%.

Further vectorization of GNSD, with the modified data structure resulted in the improved

performance as summarized by the results of Table 3 (column 4). For some of the larger test problems as much as 99% of the total execution time is spent in the subproblem; no further improvement is possible in this environment since the current implementation of the primal simplex algorithm for generalized networks is inherently scalar. Using an alternative linear programming algorithm (e.g. Karmarkar[24]) might improve the related performance of the SD method. An obvious change to GNSD would involve restructuring internal tactics, so that more emphasis is placed on the master problem. However extensive computational testing with GNSD elsewhere (see Mulvey et al. [34]) does not provide any evidence that we could reduce the number of subproblems by solving the master problems to higher accuracy. It seems that substantial savings in vectorizing GNSD can be achieved only with some radical modifications of the underlying algorithm.

Problem	GNSD Solution times (sec)			
	Without Vectorization	Compiler Vectorization	Local Software Vectorization	Global Software Vectorization
PTN30	0.206	0.191	0.186	0.145
PTN150	1.289	1.173	1.146	0.999
PTN660	9.046	8.553	8.422	7.725
STICK4	27.080	*	25.100	23.623
GROUP1ac	3.825	3.357	3.218	2.884
GROUP1ad	8.115	*	8.007	7.728
Av. ratio	1.00	0.95	0.93	0.87

* Problem not solved with this option

Table 3 : Vectorization of **GNSD** on the **CRAY X-MP/24**

5. Discussion and Conclusions

Two conclusions are immediate from the work described in this paper:

(1) The increased complexity of computer systems adds to the user the burden of specializing his algorithms. It is insufficient to concentrate research efforts solely on designing algorithms that capitalize on the problem's intrinsic characteristics. We now must concern ourselves with the underlying computer architecture.

(2) Different algorithms exhibit various degrees of parallelism. The developed vectorization framework, when applied in the context of a truncated Newton algorithm, resulted in substantial improvement in performance - as much as 80%. The same framework, when applied on a simplicial decomposition algorithm, resulted in very modest improvement in efficiency - at most 15%. Not only must we concern ourselves about which algorithm is suitable for some class of problems, but also which algorithm is more appropriate for a given computer architecture.

In an effort to extend our previous work in comparing the network specialized algorithms with a general purpose code, we solved three characteristic problems with NLPNETG and MINOS[36] both on the CRAY and a VAX 11/750. The relative performance of the two algorithms - as shown in Figure 11 - indicates that network algorithms vectorize at least as well as general purpose codes. It seems unlikely therefore that specialized NLGN algorithms will become obsolete, given their high performance ratio to general purpose codes, even on vector computers.

Finally, an attempt was made to establish the size of the problems that can be solved on different computer systems. Table 5 summarizes the results of our testing on three diverse machines: (1) VAX 11/750 minicomputer, (2) IBM 3081 large mainframe and (3) the CRAY X-MP/24. We point out two facts that are obvious from this exercise:

(1) Ultra-large NLGN can be solved on a routine basis on vector supercomputers, when sufficient care is taken to streamline the algorithms for the machine architecture.

(2) The average ratio of 17 between the performance of NLPNETG on the IBM 3081 and the

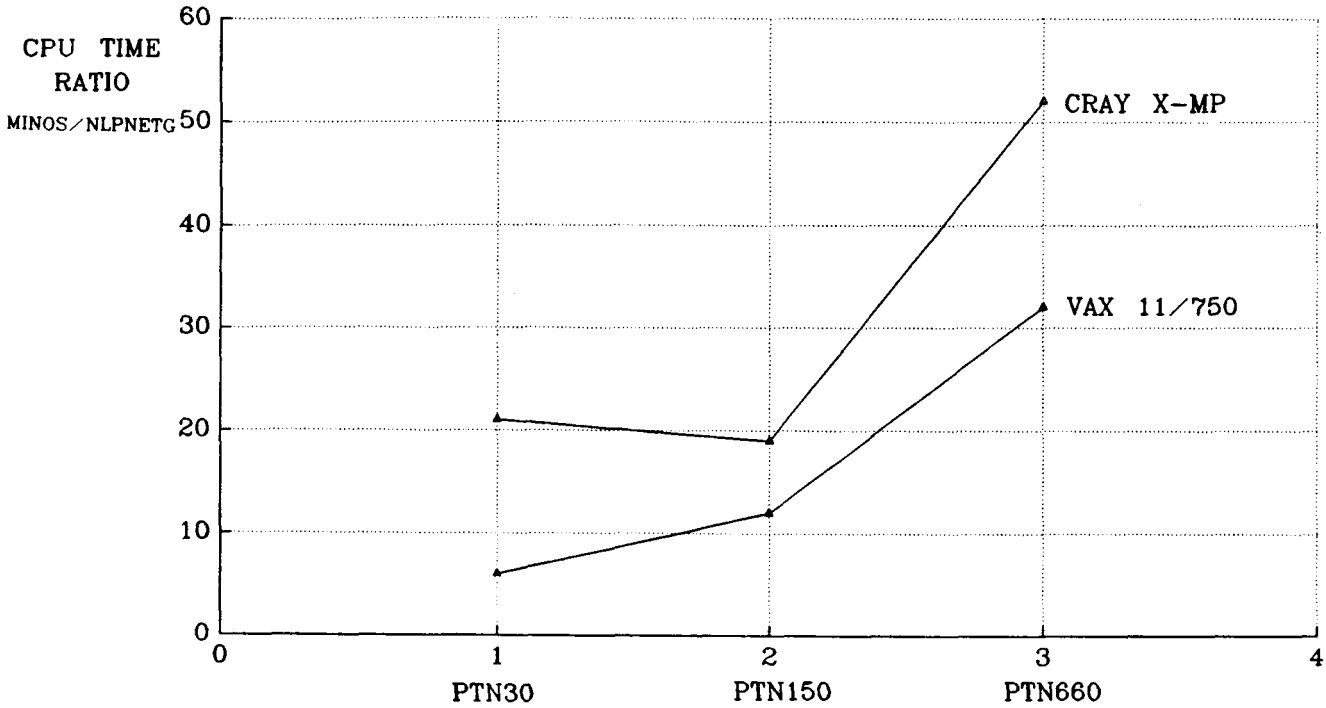


Figure 11: Performace ratio of **MINOS / NLPNETG** on **CRAY X-MP** and **VAX 11/750**

CRAY, can serve as the basis for a rudimentary comparison of nonlinear network optimization with other ares of large scale computations : Dongarra[13] reports an average ratio 13-22, for the solution of large systems of linear, dense equations and Schnedel[39] reports a ratio of 13 for seven benchmark programs for the solutions of partial differential equations. Network algorithms are obviously on the frontier of vectorizable codes.

Parallel computing opens several new areas of research for network programmers. We expect to see novel large scale applications of nonlinear network modeling. A new research topic is the development of specialized vector techniques for handling graph data structure, thus improving the performance of linear network algorithms, and with an impact on other areas of sparse matrix computations. Exploitation of the multiple processors available on the CRAY X-

Problem	NLPNETG Solution times (sec)		
	IBM 3081	VAX 11/750 (Unix)	CRAY/XMP
PTN150	1.98	23.86	0.165
PTN660	22.85	297.93	1.402
SMBANK	2.64	21.50	0.177
BIGBANK	376.74	9100.00	58.896
GROUP1ac	218.46	1652.00	4.983
GROUP1ad	1320.82	10227.00	49.454
MARK3	24.87	204.43	2.131
Average	281.19 (17)	3075.25 (184)	16.744 (1)

Table 5 : Testing **NLPNETG** on different computer systems

MP/24 is another area where further work is needed. The commercial availability of massively distributed systems presents another dimension to the problem of parallel algorithmic development.

Acknowledgment

Part of the computational work was completed while the first author was attending the Supercomputing Summer Institute sponsored by NSF and DOD, and organized by Boeing Computer Services at the University of Washington, Seattle in August 1985. The invitation of BCS to attend the Institute is acknowledged, together with helpful discussions with Mr Mike Healy and Dr Horst Simon.

References

1. *Cray-1 Computer System*, FORTRAN (CFT) Training Volume, CRAY Research Inc., 1982.

2. "Access to Supercomputers," OMB 3145-0058, National Science Foundation, Washington, D.C., August 1985.

3. D. P. Ahlfeld, R. S. Dembo, J. M. Mulvey, and S. A. Zenios, "Nonlinear Programming on Generalized Networks," Report EES-85-7, submitted for publication to Transactions on Mathematical Software, Princeton University, June 1985.

4. B.L. Buzbee and D.H. Sharp, "Perspectives on Supercomputing," *Science*, vol. 227, no. 4687, pp. 591-597, February 1985.

5. S.S. Chen, "Large-Scale and High-Speed Multiprocessor System for Scientific Applications," in *High Speed Computation*, NATO ASI Series F, vol. 7, Springer-Verlang, Germany, 1984.

6. L. Cooper and L.J. LeBlanc, "Stochastic Transportation Problems and Other Network Related Convex Problems," *Naval Research Logistics Quarterly* , 1977.

7. L. Cooper and J. Kennington, "Steady State Analysis of Nonlinear Resistive Electrical Networks Using Optimization Techniques," Technical Report IEOR 77-12, Southern Methodist University, 1977.

8. I.W. Cotton, "Technologies for Local Area Computer Networks," *Computer Networks*, vol. 4, pp. 197-208, North-Holland, 1980.

9. B. Dembart, "Vectorization using GATHER and SCATTER," ETA-TR-26, Mathematics and Modeling Technical Report, Boeing Computer Services, Seattle, April 1985.

10. R. S. Dembo and J. G. Klincewicz, "A Scaled Reduced Gradient Algorithm for Network Flow Problems with Convex Separable Costs," *Mathematical Programming Studies*, vol. 15, pp. 125-147, 1981.

11. L.C.W. Dixon and K.D. Patel, "The Place of Parallel Computing Numercial Optimization," NOC Technical Report 125, Hatfield, U.K., 1982.

12. J.J. Dongarra and A.R. Hinds, "Unrolling Loops in FORTRAN," *Software - Practice and Experience* , vol. 9, pp. 219-226, 1979.

13. J.J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Technical Memorandum No.23, Argonne National Laboratory, Illinois, April 1985.

14. A. Drud, S.A. Zenios, and J.M. Mulvey, "Blancing Some Large Social Accounting Matrices with Nonlinear Network Programming," in preparation, Development Research Department, The World Bank, Wahington, D.C., January, 1986.

15. I.S. Duff, "The Used of Supercomputers in Europe," CSS-161, Computer Science and Systems Division, AERE Harwell, Oxfordshire, September 1984.

16. M. Florian and S. Nguyen, "A Method for Computing Network Equilibrium with Elastic Demands," *Transportation Science*, vol. 8, pp. 321-332, 1974.

17. M. J. Flynn, "Some Computer Organizations and their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948-960, 1972.

18. P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization,* Academic Press, London, 1981.

19. R. V. Helgason and J. L. Kennington, "An Efficient Specialization of the Convex Simplex Method for Nonlinear Network Flow Problems," Technical Report, IEOR 77017, Southern Methodist University, 1978.

20. R. Hockney and C. Jeeshope, *Parallel Computers,* Adam Hilger Ltd., Bristol, England, 1981.

21. B. von Hohenbalken, "Simplicial Decomposition in Nonlinear Programming Algorithms," *Mathematical Programming*, vol. 13, pp. 49-68, 1977.

22. K. Hwang, *Supercomputers: Design and Applications,* IEEE Computer Society, Maryland, 1984.

23. M.H. Kalos, "The NYU Ultracomputer," Ultracomputer Note 48, Courant Institute of Mathematical Sciences, New York University, N. York, April 1983.

24. N. Karmarkar, "A New Polynomial Algorithm for Linear Programming," *Combinatorica*, vol. 4, pp. 373-395, 1984.

25. M.J. Kascic, Jr., "A Performance Survey of the CYBER 205," in *High Speed Computation*, NATO ASI Series F, vol. 7, Springer-Verlang, Germany, 1984.

26. J. S. Kowalik, *High Speed Computations*, NATO ASI Series, Computer and System Sciences, 7, Springler-Verlang, 1984.

27. J.L. Larson, "Multitasking on the CRAY X-MP/2 Multiprocessor," *Computer*, July 1984.

28. L.S. Lasdon and A.D. Warren, "A Survey of Nonlinear Programming Applications," *Operations Research*, vol. 28, pp. 34-50, 1980.

29. L.S. Lasdon, "Adaptation of Optimal Acoustic Antenna Array Design Code to the CRAY 1-M," (manuscript), 1985.

30. C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh, "Basic Linear Algebra Subprograms (BLAS) for Fortran Usage," *ACM Transactions on Mathematical Software*, vol. 5, pp. 308-323, 1979.

31. F.A. Lootsma, "Parallel Unconstrained Optimization Methods," Report 84-30, Department of Mathematics and Informatics, Delft University of Technology, Netherlands, 1984.

32. K. Miura and K. Uchida, "FACOM Vector Processor VP-100/VP-200," in *High Speed Computation*, NATO ASI Series F, vol. 7, Springer-Verlang, Germany, 1984.

33. J. M. Mulvey, "Nonlinear Network Models in Finance," *Advances in Mathematical Programming and Financial Planning*, JAI Press, 1985.

34. J. M. Mulvey, S. A. Zenios, and D. P. Ahlfeld, "Simplicial Decomposition for Convex Generalized Networks," Report EES-85-8, Princeton University, 1985.

35. J. M. Mulvey and S. A. Zenios, "Integrated Risk/Cost Planning Models for the U.S. Air Traffic System," Report EES-85-9, Princeton University, June 1985.

36. J.M. Mulvey and S.A. Zenios, "Managing Software Systems in a Rapidly Changing Hardware Environment," Report EES-86-2,, Princeton University, 1986.

37. B. A. Murtagh and M. A. Saunders, "MINOS User's Guide," Report SOL 77-9, Department of Operations Research, Stanford University, California, 1977.

38. K. W. Neves, "Impact of Changing Architecture," Report ETA-TR-28 , Engineering Technology Applications, Boeing Computer Services, Seattle, 1895.

39. R. E. Rosenthal, "A Nonlinear Network Flow Algorithm for Maximization of Benefits in a Hydroelectric Power System," *Operations Research*, vol. 29, pp. 763-786, 1981.

40. U. Schendel, *Introduction to Numerical Methods for Parallel Computers*, Ellis Horwood, Chichester, U.K., 1984.

41. R.B. Schnabel, *Parallel Computing in Optimization*, NATO ASI on Computational Mathematical Programming, F.R. Germany , 1984.

42. C.L. Seitz, "The Cosmic Cube," *Communications of the ACM*, vol. 28, no. 1, pp. 22-33, January 1985.

43. B.J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Real Time Signal Processing IV, Proceedings of SPIE*, pp. 241-248, The International Society of Optical Engineering, 1981.

44. J. Warlton, "Supercomputers: Past - Present - Future," in *Proceedings of the BCS Supercomputer Summer Institute*, University of Washington, Seattle, Aug., 1985.